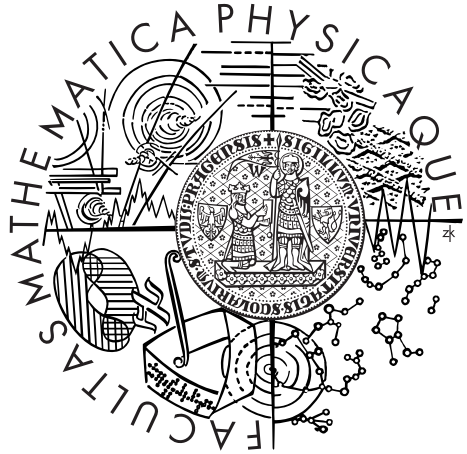
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Tomáš Jakl

Arimaa challenge – comparission study of MCTS versus alpha-beta methods

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Vladan Majerech, Dr.
Study programme: Computer Science
Specialization: General Computer Science

Prague 2011

I would like to thank to my family for their support, and to Jakub Slavík and Jan Vaněček for letting me stay in their flat at the end of the work. I would also like to thank to my supervisor for very useful and relevant comments he gave me.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date August 5, 2011 Tomáš Jakl

Title: Arimaa challenge – comparission study of MCTS versus alpha-beta methods

Author: Tomáš Jakl

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Vladan Majerech, Dr., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In the world of chess programming the most successful algorithm for game tree search is considered AlphaBeta search, however in game of Go it is Monte Carlo Tree Search. The game of Arimaa has similarities with both Go and Chess, but there has been no successful program using Monte Carlo Tree Search so far. The main goal of this thesis is to compare capabilities given by Monte Carlo Tree Search algorithm and AlphaBeta search, both having the same evaluation function, in the game of Arimaa.

Keywords: Arimaa, Monte Carlo Tree Search, alpha-beta, abstract strategy game

Název práce: Arimaa challenge – srovnávací studie metod MCTS a alfa-beta

Autor: Tomáš Jakl

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Vladan Majerech, Dr., Katedra teoretické informatiky a matematické logiky

Abstrakt: Ve světě šachových programů je považováno AlphaBeta prohledávání za nejvíce úspěšné, na druhou stranu ve světě Go je to Monte Carlo Tree Search. Hra Arimaa je podobná jak Go tak šachům, ale zatím se nestalo, že by se objevil úspěšný hrající program používající Monte Carlo Tree Search. Hlavním úkolem této práce je porovnat schopnosti Monte Carlo Tree Search a AlphaBeta prohledávání, když oba algoritmy budou používat stejnou ohodnocovací funkci.

Klíčová slova: Arimaa, Monte Carlo Tree Search, alpha-beta, abstraktní strategická hra

Contents

1. Introduction

After first computers were created there were always an intention to compete human mind in many occasions. First significant result has occurred in 1997. IBM's employees succeeded by constructing a computer named Deep blue built with just one purpose, to defeat Garry Kasparov, the best human player in the game of chess. The core of an algorithm used in Deep blue was AlphaBeta search.

Because Kasparov was defeated, new challenges has come. Defeat men in Go. Until now, there were no significant success on standart 19×19 board. But many useful new ways of playing games were invented. In 2006 Monte Carlo methods were used to defeat all other computer players of Go. After that day all successful programs were using Monte Carlo methods [11].

In the world of chess programming the most successful algorithm for game tree search is still considered AlphaBeta search, however in game of Go it is Monte Carlo Tree Search. Arimaa has similarities with both Go and Chess, but there has been no successful program using Monte Carlo Tree Search so far.

On the 1st February 2010 in Arimaa forum, an interesting question was asked, how would two bots – one using Monte Carlo Tree Search and the second using AlphaBeta search – compete if they both would use the same evaluation function [3].

We will introduce rules of the game of Arimaa and describe two algorithms MCTS[1] and the AlphaBeta search with various extensions.

1.1 Terminology

Game tree for arbitrary game is tree with starting positions as root and with children of nodes as all possible consequent positions.

Evaluation function is function which estimates value of given position of the game. It can be used for example to compare which of two given positions is better.

Minimax tree for two player game is game tree limited to some depth with added values in all nodes. Values are defined recursively. In a leaf of the tree the value is calculated by evaluation function. In nonleaf node the value is defined as the best value from nodes children from point of view of nodes active player.

Principal variation is best sequence of moves for actual player leading from root of the tree to leaf in minimax tree if we presume both players play their best.

Transposition is a position which can be reached by more than one sequence of moves.

Branching factor of a node is number of children that node has. Usually branching factor of a tree is an average branching factor of all its nodes.

[1] Monte Carlo Tree Search

Game bot is a program playing the game.

1.2 The Game of Arimaa

"Even simple rules can lead to interesting games."
— Omar Syed

The game of Arimaa belongs to younger games. It is carefully designed to be hard to play for computers and easy to play for humans. Arimaa's creator, Omar Syed says that Kasparov was not oversmarted but overcomputed by Deep blue. This motivated him to create game with such properties.

In order to make the game easier for humans and harder for computers he brought an interesting idea: "The pieces should have very simple movements, but the players should be able to move more than one piece in each turn" [3].

Another weakening of computers is achieved by having rules that embarrasses methods well known from Chess such as game-ending tables or opening tables. It should be also significantly harder to efficiently decide which of two given position is better [3].

In spite of the fact that in Arimaa we have six different kinds of pieces, almost all of them moves the same way which makes the rules of the game much easier for human in comparison to chess, furthermore the rules of Arimaa are so simple that even small kids can play it.

1.3 Rules of the game

Arimaa is a two-player zero-sum game with perfect information. It is designed to be possible to play it using the board and pieces from chess set. The starting player has Gold color, second is Silver. In the first turn Gold and then Silver player each place all their pieces into first two (for Gold) or last two (for Silver) lines of the board any way they consider appropriate. Piece set consist of eight Rabbits, two Cats, two Dogs, two Horses, Camel and Elephant in order from weakest to strongest.

Players are taking turns starting with Gold. In each turn a player makes move, which consist of one to four steps. Player may *pass* and do not use remaining steps. After move ends, the position of the board must be different from the position before move started and a player is not allowed to play the same position for third time.

A piece is touching another piece if it is staying on a neighbouring square of that piece. We say a piece is lonely if it is not touching another piece sharing color with it. A lonely piece is frozen if it is touching an opponents stronger piece. A catchable piece by some stronger piece is an opponents piece touching that stronger piece. Adjacent or neighbouring squares are those squares lying in one position to left, right, front or backwards.

To make step a player chooses one of its non frozen pieces and move it to one of the free adjacent squares, with one exception – rabbits cannot step backwards. Instead of making a simple step player can provide pull or push.

In pulling the player choses his non frozen piece and a piece catchable by it. Normal step by the player's piece is performed and then the catchable piece is moved to the stepped's piece previous location. Pushing is a kind of opposite to pulling. Player on turn chooses a catchable piece and his stronger piece touching it. Then by the player's choice the opponent's catchable piece is moved to its free adjacent square and the stronger non frozen piece touching it is moved to the catchable piece's former location. Push and pull counts as 2 steps and can not be combined together.

On the board are four special squares called traps in positions `c3`, `c6`, `f3` and `f6`. A lonely piece standing in a trap square after a step is immediately removed from the board.

Following end-game conditions are checked at the end of each turn:

1. Scoring *goal:* one of the rabbits has reached goalline (the furthest row from the starting rows). The player whose rabbits reached the goalline wins. In the artificial case when rabbits of both players reached the goalline the player making the move wins.

2. *elimination:* one of the players has no rabbit left. The player having a rabbit wins. In case where no rabbit remains, the player making the move wins.

At the start of a turn an existence of a legal step is checked. If such step does not exist, the player to move loses due to *immobilisation*.

Making illegal move leads to losing the game. This cannot happen to human player using modern computer interface, but it could happen to buggy computer programs or in the game played on the real board. The games played using computer interface also can end by *timeout* when a player to make step do not make it at time.

There are special rules to define a game result even for games taking too long time, but these rules will not be important for us.

More informations about rules can be found in [3].

1.4 Comparison to Go and Chess

Because Arimaa is played with full chess set it is very natural to ask about similarities with the game of Chess. In Chess and in Arimaa it is so easy to ruin good position with just one bad move. For example stepping out of trap an therefore let another piece to be trapped or unfreezing rabbit near goalline. Unlike in Chess starting position is not predefined and there are about 4.207×10^{15} different possible openings which makes it hard for Arimaa bot programmer to use any kind of opening tables [2].

In Arimaa it is also very hard to build a good static evaluation function [16]. Event two current best players do not agree in evaluation of camel for cat with horse exchange, which wildly depends on the position of other pieces. The simplest example of our lack of knowledge is evaluation of initial rabbit for nonrabbit exchanges. Some of the top players evaluate initial exchange of two rabbits for one horse almost equal while others prefer a cat for two rabbits.

Building good evaluation function is very hard in game of Go also, because it requires a lot of local searching to find territories and determine their potential owners. On the other hand destroying good position by making wrong step is in Go a lot harder.

In the game of Go if we omit filling eyes, then any random playing sequence leads to end, which is not so easily achievable in Arimaa and Chess. Christ Cox showed in his work that in Arimaa branching factor of average position is around 20,000. With comparison in Chess it is only 35 and in Go 200 [2].

1.5 Challenge

Omar Syed decided to left a prize 10,000 USD for programmer or group of programmers who develop Arimaa playing program which win Arimaa Computer Championship and then defeats three chosen top human players before the year 2020. Omar Syed believes that this motivation will help further improvements in area of AI game programming. So far computers were not even close to defeat one of the chosen human players [3] [4].

1.6 Object of research

Tomáš Kozelek has shown in his work, that building Arimaa playing program using MCTS is possible. In this work we will focus on comparing capabilities and perspectives to the future given by AlphaBeta search and MCTS in game of Arimaa.

The main part of this work is to develop Arimaa playing program and try to answer the following questions:

1. Is MCTS competitive to alpha beta search at all?
2. Is MCTS more promising engine than AlphaBeta search in the future with increasing number of cpus?
3. How important is evaluation function in MCTS compared to AlphaBeta?

2. Algorithms

In this chapter we introduce two algorithms for searching minimax trees – AlphaBeta search and Monte Carlo Tree Search (MCTS). Both algorithms became successful and dominating in some field, AlphaBeta search in game of Chess and MCTS in game of Go.

The basic algorithm to search game tree is Minimax search. It is searching minimax tree in depth first search order and therefore the search do not ends until all nodes at certain depth are explored. It is quite noneffective with huge branching factor which Arimaa, Chess or Go have.

2.1 Description of the AlphaBeta search

AlphaBeta search has grown in popularity in game of Chess. Until now all successful arimaa bots were using it with various enchants. AlphaBeta search is natural optimisation of Minimax search. The main purpose of algorithm is to reduce number of branches and nodes to be visited.

During the search we are trimming bounds (window) of the best minimax value. The pseudocode of the algorithm is shown in 2.1.

```
alphabeta (node, depth, alpha, beta):
    if  depth = 0 or is_terminal(node):
        return evaluate(node)
    if  is_maximizing(node):
        for each child of node
            score = alphabeta(child, depth-1, alpha, beta)
            alpha = max(alpha, score)

            if beta ≤ alpha: break  # Beta cut-off
        return alpha
    else:
        for each child of node:
            score = alphabeta(child, depth-1, alpha, beta)
            beta = min(beta, score)

            if beta ≤ alpha: break  # Alpha cut-off
        return beta
```

Algorithm 2.1: Pseudocode of the AlphaBeta search

In maximizing nodes we are improving lower estimate (alpha) of the minimax value and in minimizing nodes the upper estimate (beta). If a value from a child forces these bounds to meet we know that better a approximation of this node is not possible and so we return our estimate, we say we pruned/cut-off the search in node.

It can be proven that if AlphaBeta finds solution for depth n, then it is the best solution in MiniMax tree for depth n [5]. In optimal case instead of examining $\mathcal{O}(b^d)$ nodes is examined only $\mathcal{O}(b^{d/2})$ nodes, where b is branching factor of the game and d is searched depth [6].

The most time is spared if the pruning children of nodes are listed as first. However in the worst case if children are sorted in opposite order to the optimal, the whole minimax tree is searched. Consequently it is very important to have nodes ordered well.

2.2 Description of the Monte Carlo Tree search

Monte Carlo methods were formerly used for approximation of mathematical, physical, biological and others processes where full calculation would be difficult or even impossible. For example in mathematics it is used for numeric integration or estimate the value of π. In games without perfect information, such as poker, backgammon, or scrabble, using randomness has shown to be beneficial too [9, 10].

In game of Go was always a huge troublesome to build efficient evaluation function. Unlike poker or backgammon, Go is a game with perfect information and hence it is not so natural to use Monte Carlo methods. The first attempts to use random approach as evaluating approximation of the Go position were in 1993. We present generalised Bernd's algorithm [8, 1]:

1. Play random game from given position to the end, with one exception, choose only steps not filling eyes. At the end of simulation count in the result of simulation for the first step played.

2. If there is time left go to 1.

3. Choose move with highest ratio between number times the move won when it was played and the number of times it was played.

However Bernd showed that after some time his algorithm indicates no further improvement. It was necessary to realize that the main problem is that algorithm searches all branches with the same probability. It should give more attention to more promising branches. The question how to find fine balancing between exploration and exploitation is classical mathematical Multi-armed Bandit problem.

2.2.1 Bandit Problem

A K-armed bandit, is slot machine with K arms. When arm is drawn it produces reward. Distribution of each arm reward is independent on other arms and previous draws of this arm. The task is to choose best strategy to maximize sum of reward through iterative plays [11, 12].

In [12] is presented three UCB1 algorithm (where UCB stands for Upper Confidence Bounds) for Bandit Problem:

1. Play each arm of the bandit once.

2. Play arm maximizing the formula $\overline{X}_i + \sqrt{\frac{2 \log n}{n_i}}$, where $\overline{X}_i$ is average value of the arm i, n is number of games that were played by parent of the i and n_i is number of games played with arm i.

2.2.2 UCT algorithm

UCT is abbrevation for Upper Confidence bound to Trees. Shortly described it is the UCB algorithm applied to minimax trees. The main idea is to consider each node of minimax tree as multiarmed bandit problem and each child of the node as independent arm [11].

2.2.3 MCTS

Monte Carlo Tree search is generic best-first-search algorithm applied to trees which uses a random simulation as evaluation scheme. It consist of four steps which are repeated as long as there is time left. The algorithm starts with a tree containing only a root node representing starting position [7, 1].

The four mentioned steps are explained in the following list:

1. *Selection:* Until a leaf node is reached a child of node is recursively selected by using best-first manner. The node selection is usually handled by UCT algorithm.

2. *Expansion:* The selected leaf's children are generated.

3. *Simulation* (also called *playout*): A random game is played from given position to certain depth.

4. *Backpropagation:* The result of simulated games is stored into all nodes visited during this iteration of algorithm in the Selection phase.

Classical approach of the Simulation phase in Go bot programming is to generate only steps not filling eyes and to stop playout when the end of game is reached (which is not possible in Arimaa) [11].

The result of the Simulation phase is computed from the last position of the simulation by a evaluation function. The probabilistic attitude of UCT/UCB1 requires the value of evaluation to be scaled in interval $[-1, 1]$ [1].

A pseudocode of the full algorithm is shown in 2.2 (inspired by [11]).

```
playOneIteration():
    node[0] = rootNode
    i = 0
    while is_not_leaf(node[i]):
        node[i+1] = descendByUCB1(node[i])
        i = i + 1
```

```
    expandNode(node[i])
    node[i].value = getValueByMonteCarlo(node[i])

    for j in {0, ..., i-1}:
        updateValue(node[j], node[i].value)

while is_time():
    playOneIteration()

print descendByUCB1(rootNode)
```

Algorithm 2.2: Pseudocode of the MonteCarlo Tree Search

We modified Kozelek's UCB exploration formula to form:

$$\overline{X}_i + c\sqrt{\frac{log\ n}{n_i}} + \frac{h_i}{n_i} + \frac{hh_i}{n_i}$$

where $\overline{X}_i + c\sqrt{\frac{log\ n}{n_i}}$ is a generalised UCB1 formula, h_i is heuristics evaluated for step leading to this position, and hh_i is history heuristic value.

One of the negatives of UCT algorithm is that UCB1 formula itself presumes for the involved random variables to be identically distributed and independent, which is not true in UCT [11]. It was shown that the probability of selecting the correct move quickly converges to 100% [18].

3. Used optimisations in search engines

In order to write a strong playing program in given game, it is necessary to enhance chosen algorithm with various extensions. Tuning a set of used extensions and their parameters is one of the key factors in bot development. In this chapter we will describe the most common extensions for AlphaBeta or MCTS algorithms. In this chapter we will describe optimisations we tried to implement in our game engines.

3.1 AlphaBeta

Implementing listed extensions has long background in game of Chess. Usually in Arimaa, some combination of well approved enhances is used.

3.1.1 Transposition Tables

If we look at the game tree of Arimaa, there is so many repetitions in nodes of the tree for almost every position on the board. The Transposition Tables are used to reduce the number of repetitions.

In Arimaa thanks to Step-Turn approach of the game, handling transpositions is a little harder than it is in Chess. We have to worry about number of steps in turn that were made on top of the classical transpositions handling.

When Transposition Tables are used, every time we are about to explore some node, we look at first to the Transposition Tables. If a entry of the same transposition occurs in previous searches, depending on its searched depth we do:

- If the entry's search is shallower than we need, we prefer steps from Principal Variation of given entry in succeeding search and adjust also our bounds with its.
- However if the entry has been searched to equal or greater depth we stop searching of this node and use a full Principal Variation and a score of that entry as result of exploring our node.

When an arbitrary position is fully explored, its best value and a corresponding Principal Variation are saved to the Transposition table [2].

Due to limited space, a policy for reusing space is needed. Moreover, results of deeper searches are usually considered as a good approximation, but because of the evaluation instability using Transposition Tables could lead to different results. Order of nodes expansion could therefore affect the result.

3.1.2 Iterative Deepening Framework

Because there is no way how to determine how long the AlphaBeta search to given depth will take, we need to find some time management tool. This is a method in which we are iteratively starting new deeper and deeper searches. Thanks to the exponential growth of the minimax tree, we know that with increasing search depth by one level is the number of nodes newly searched asymptotically bigger than the number of nodes explored in previous shallower searches.

Because we can sort steps using information gained from previous shallower searches Iterative deepening used with other optimisation methods such as History Heuristic or Transposition Tables often cause a time save if we compare it to just searching to a maximal possible depth [2].

3.1.3 Aspiration Windows

Iterative Deepening itself can be further improved. The AlphaBeta search normally starts with (α, β) bounds (window) set to $(-\infty, \infty)$. When an Aspiration Window is set, we start a search with $(prev - window, prev + window)$ window instead, where $prev$ is the value from previous shallower search and $window$ is certain predefined constant. If the result of the search fall outside the window, the new search with wider window must be performed.

With narrower windows, more pruning during AlphaBeta search should occur and therefore the search to certain depth should end much earlier. However if a value of a search often miss a window, the need for repetitions may actually linger the search [19].

3.1.4 Move ordering

The following methods change the order in which branches are selected and then inspected. It is very important for the AlphaBeta search to have nodes well ordered, because earlier we find a pruning child of a node the shorter time we spend in it.

History heuristics

The main idea behind this extension is that if some step is good enough to cause so many pruning anywhere in a search tree and if it is valid in given position it could be also good here.

A table of scores for every combination of player, piece, location and direction is stored into memory. During the AlphaBeta search we increase a score of an element every time a step with such combination causes pruning or became a child with the best score for an actually searched node. It is believed that the deeper cut-off happens the more relevant it is and therefore the score is incremented by d^2 or 2^d where d is an actual searched depth.

During searching nodes of the minimax tree are sorted by score from the History Heuristic table in decreasing order [2].

Killer moves

When a step prunes other branches in some node it is very natural to ask, if the same step could cause pruning in another branch and the same depth of the tree.

Therefore the last pruning step from the same level is tried right after Transposition Table's Principal Variation. To go even further we take two last steps causing pruning to be preferred in the search. As Cox found out, three or more Killer moves would not help [2].

Null move

The main idea is that if a player is in bad position and if we skip its opponent's turn and the position still cannot be strongly improved, then there is no chance for this position to be good.

During search in a new-turn nodes, either the whole opponent's turn is skipped (which is called Null move) or normal search is performed. In implementation the Null move is searched as first child of a node and it should shorten the time until cutoff occurs in really bad positions. Performing two Null moves in row or in the root node is forbidden [2].

3.2 Monte Carlo Tree Search

In AlphaBeta search we made great effort in sorting nodes of the searched tree properly. In MCTS we need to use optimisations which helps us to gain more and better informations from each iteration of algorithm on top of that.

We chosed to implement only heuristic Tomáš Kozelek described in [1] as useful.

3.2.1 Transposition Tables

The motivation is the same as it is in the AlphaBeta algorithm (see 3.1.1). However in MCTS we are increasingly building game tree instead of just exploring branches to some depth. A natural use of Transposition Table for MCTS is to share statistics for the same transpositions in built game tree.

To do so we bind nodes considered the same to one. Bound nodes share their children nodes, visit count and score statistic. We say that two nodes are the same if they are in the same depth in minimax tree and if they represent the same transposition. The game tree became Directed Acyclic Graph.

In implementation during the computation we keep table of all transpositions and when any node is expanded we bind it to transposition in table if exists.

In Kozelek's work is regarded to be dangerous to bound visit count and score statistic for children nodes [1]. Nevertheless we believe that if some step leads us to a position which is proven to be not worth trying in some branch the same stands for all its transpositions.

3.2.2 Progressive bias

Progressive bias technique is a nice way how to combine offline learned informations with online learned informations. The more we go through a node the importance of offline learned information goes down and the importance of online learned information became superior.

To do so $\frac{H_B}{n_i}$ is added to the UCB formula. Where H_B is progressive bias coefficient computed by step-evaluation function [7].

In Arimaa such step-evaluation function should appreciate steps with Elephant moving, killing an opponent's piece, around previous steps, which are pushing or pulling or making goal. Such function should also handicap a player's own piece sacrifice or inverse steps [1].

3.2.3 History heuristics

Tomáš Kozelek brought this optimisation used in AlphaBeta search to MCTS. Similarly to AlphaBeta's approach, it is used to share informations gained in one branch of the searched tree with other branches.

In order to rate steps, we keep statistics (score and visit count) for each combination of player, piece, location and direction.

These statistics are updated during the Backpropagation part of the MCTS. For each updated node we also update the statistics of the step leading to that node and used in the Selection part of MCTS. The $+\frac{hh_i}{n_i}$ expression is added to UCB1 formula. Where hh_i is history heuristics coefficient representing the mean value of step leading to is child of a searched node score. Original idea is described in Kozelek's work [1].

3.2.4 Best-of-N

In random simulations, we may want to sacrifice true randomness for gaining more objective results from playouts [13]. Instead of generating single random step, N random steps are generated instead and a step with the best value given by the same step-evaluation function as was used in Progressive bias is chosen.

As a consequence, the number of playouts decreases, but the quality of information learned in playouts improve and therefore the strength of the program improve as well.

3.2.5 Children caching

Kozelek introduced natural method how to decrease an amount of time spent in node during selection part of the MCTS.

After some number of visits of a node its children caching is switched on. Then a few best children are chosen and cached and every time the selection part of the MCTS goes through this node it chooses a descend node from earlier cached children. After some time it is necessary to discard and fill cache again [1].

Using this optimisations should improve the speed of algorithm without negative impact on quality.

3.2.6 Maturity threshold

Is another technique how to shorten time used in node selection. It helps to reduce the size of a MCTS tree as well. We expand only those nodes which had at least $threshold\_maternity + depth\_of(node)$ visit count.

3.2.7 Virtual visits

During node expansion we initialise new node with v virtual visits. This small change increases the power of the algorithm significantly. Kozelek experimentally determined the best performance of algorithm for $v \in [4, 5]$ [1].

3.3 Independent optimisations

The following extensions are necessarily needed in every successful Arimaa bot and they do not depend on used algorithm.

3.3.1 Bitboards

As Arimaa could be played with standard chess set it is natural to adopt well known chess board representation called Bitboards.

We represent board as twelve 64bit numbers, one for each combination of player and piece kind. The ith bit of certain piece and player combination is set to `1` if on ith square (counting from `a1` to `h8` by lines) the corresponding piece stands, otherwise the bit is set to `0`. For example if a Golds Dog is on `b2` the 10th bit in the Golds Dogs number is set to `1`.

One may say that the Bitboards suits even better for Arimaa than for Chess. For instance computing simple steps is for almost all pieces just taking precomputed bit number of all adjacent squares for his coordinates and `AND`ing it with 64bit number representing all empty squares on the board [17].

3.3.2 Zobrist keys

Zobrist keys are one of the optimisation methods well known from world of chess. With using Transposition Table comes the need for fast determination mechanism if two positions are the same, and for effective generating almost unique hash keys for storing transpositions into hash table.

For every tripes $piece \times player \times location$ the random 64bit number is generated and stored to array as `zobrist[piece][player][location]`. A hash value is then computed

by taking for each piece on board the corresponding number from the precomputed `zobrist` array and `XOR`ing all those numbers together.

Calculating a hash value for a board is shown in pseudocode 3.1.

```
zorbist_hash (board):
    key = 0x0000000000000000        # key of empty board
    for p in pieces_of(board):
        hash = zobrist[piece(p)][controller(p)][location(p)]
        key = key XOR hash

    return key
```

Algorithm 3.1: Computing hash value from zobrist keys

Computing such function every time the hash value is needed would not be efficient at all. The value can be easily updated after a step is made by `XOR`ing a previous board's hash value with:

```
zobrist[piece][player][from] XOR zobrist[piece][player][to]
```

Where `from` and `to` represents from-to location change by making the step. Or for push/pull steps by doing the same simultaneously with both a pushing/pulling piece and a pushed/pulled piece.

Cox showed that probability of collision for two zobrist hashes in typical search is less than 2.2×10^{-5} [2].

4. Implementation

We developed set of libraries for play Arimaa to be used with both MCTS and AlphaBeta algorithm using haskell as programming language. On top of those libraries we built mentioned algorithms. The critical parts like bit operations and evaluate function were written in C.

Haskell differs from other common used languages whith its laziness. Which means that only values that are truly needed for computation are evaluated. For example we may define infinite list of prime numbers and then ask for third element of this list which causes only the first three prime numbers to be computed.

This paradigm is applied naturaly in the AlphaBeta search. When we are exploring some node and in one of the first child's the cut-off occurs we spared time not generating others.

In Haskell it is possible to program in much higher level than it is in most other programming languages. However it is also harder to reason about performance. We are pretty sure that more experienced Haskell programmer would write both engines more efficiently.

Parameter tuning

We made huge effort to keep our program rather simple and modular as much as possible. Therefore one can switch on or off a lot of mentioned search extension. In step generator we let the possibility to generate pass step switched off by default.

Aspiration window gave sometimes strange results so we left it unused by default. Using history heuristic in Alpha Beta program tends to decrease quality of our program and we believe that causing list of steps from given position to be evaluated is limiting in comparison to have them evaluated lazily. However it is known that importance of history heuristic grows when the depth increase and hence for mor efficient programs using history heuristic is much more important [6].

Our AlphaBeta algorithm lack of standart Quiescence[1] search extension with Trap control or the Goal check[2]. We believe that both mentioned extensions could be somehow included in MCTS.

4.1 Parallelization

Creating parallel Monte Carlo Tree Search bot is much more natural than creating parallel AlphaBeta searching bot. For each CPU one thread is created and there is only one shared

[1] is an important extension to AlphaBeta search, before evaluating leafs the tactical search is performed looking for easy trapping or scoring goal.

[2] is an important extension of evaluation function or Quiescence search. Its only purpose is to check whether any player can score goal in actual/next turn.

tree for all of them. Haskell gave us easy threading and simple data structure locking capabilities thanks to standart MVar (synchronizing structure).

In a parallelized AlphaBeta search we start one thread per CPU. Every iteration of Iterative deepening framework creates a shared queue filled with all possible steps from starting position. Then each thread is taking steps eagerly from queue and updating the window and killer moves.

4.2 Evaluation function

We wrote evaluation function completely in C to be as fast as possible. It consist of hand tuned material and position evaluation, simple bonuses for having stronger piece advantage, controlling trap, freezing opponents piece and possibility to move oponents weaker piece.

We used a weak one, but the code is ready for easy replacement. Our evaluation function does not include Goal Check, less important is its lack of frame, elephant blockade or hostage detection. Normally there are no incentives in evaluation to rotate pieces to maintain these patterns while improving our options.

Well playing arimaa program needs evaluation function addressing all these issues.

5. Methodology

At the begging of the work we decided to follow the consequent schedule:

1. Study possible and most used method used in Arimaa, Chess and Go bot development.
2. Develop basic versions of bots with plain versions of AlphaBeta search and Monte Carlo Tree Search. However a work on bots is never ending so we will try to develop two comparable bots and try to measure how changes in settings affect their win:loss rate.
3. Improve both programs with enhances learned in first part of the process and tune variables of those enhances in order to improve the quality of both programs as much as possible.
4. Compare those engines playing offline matches and analyse the meaning of obtained results.

In order to obtain as much informations as possible we decided to run tests consisting of these parameters:

1. Test bots and enhances they uses by playing games with 3, 10, or 30 seconds time limits per turn to test how different time limits affects their efficiency.
2. Playing games with Transposition Tables' size limited to 100MB, 200MB, or 400MB.
3. Playing engines using one, two, four, or eight cores.

6. Results

We ran our tests on machines 12 machines with Intel® Core™ i7 CPU 920 @ 2.67GHz and 14 machines with Intel® Core™2 Quad CPU Q9550 @ 2.83GHz. In all general test cases, bots had 200MB of memory and time limit 3, 10 or 30 seconds per turn. The logs from tests are saved on attached CD.

In each experiment we performed around 400 tests. Where the around means that more than 420 tests started, but only result from tests ending in time were counted. In all figures the solid curve named `full` represents the number of games in percentages in which the MCTS using all extensions defeated the AlphaBeta search bot also with all extensions enabled.

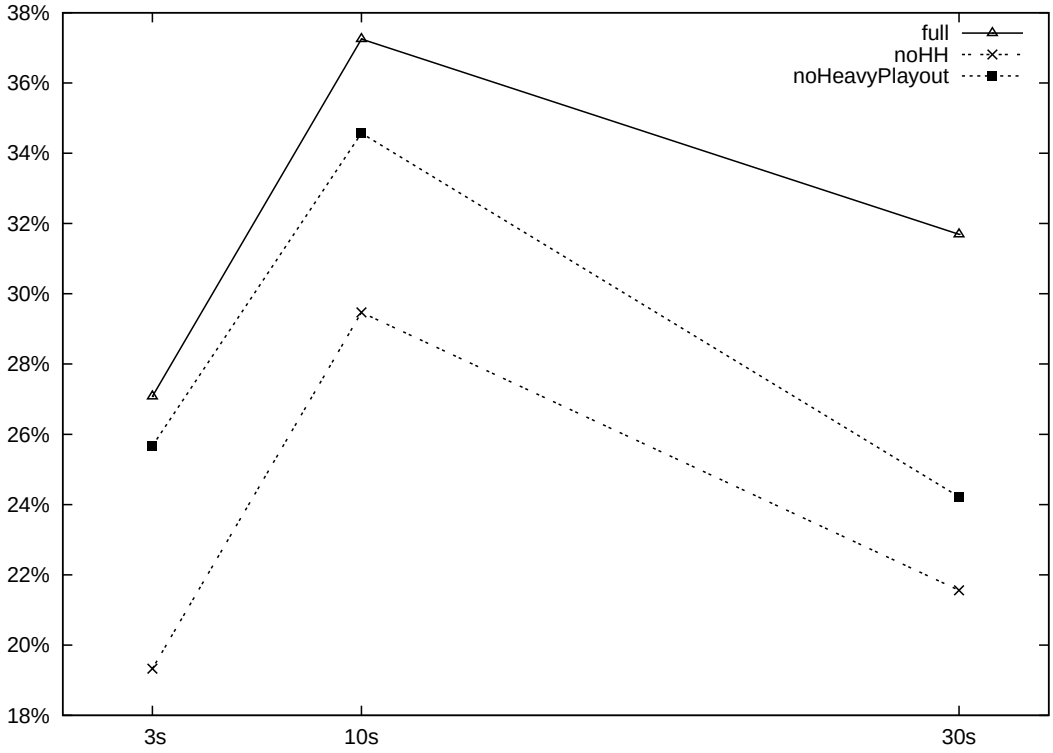


Figure 6.1: MCTS optimisations' performance

In Figure 6.1 is shown how disabling some MCTS's optimisation affects the winning ratio between algorithms. The noHH stands for switching off History heuristic in MCTS, and noHeavyPlayout stands for switching off Best-of-N optimisation. It seems from the graph that we achieved constant improvement just by switching History heuristics on. On the other hand, the importance of using Heavy Playouts increases with bigger turn limit.

In Figure 6.2 we can see how switching off Transposition Tables (as `noTT` in graph) or changing evaluation function to function used in bot_Fairy (`EVAL=fairy` in graph) changed the winning ratio between our two bots.

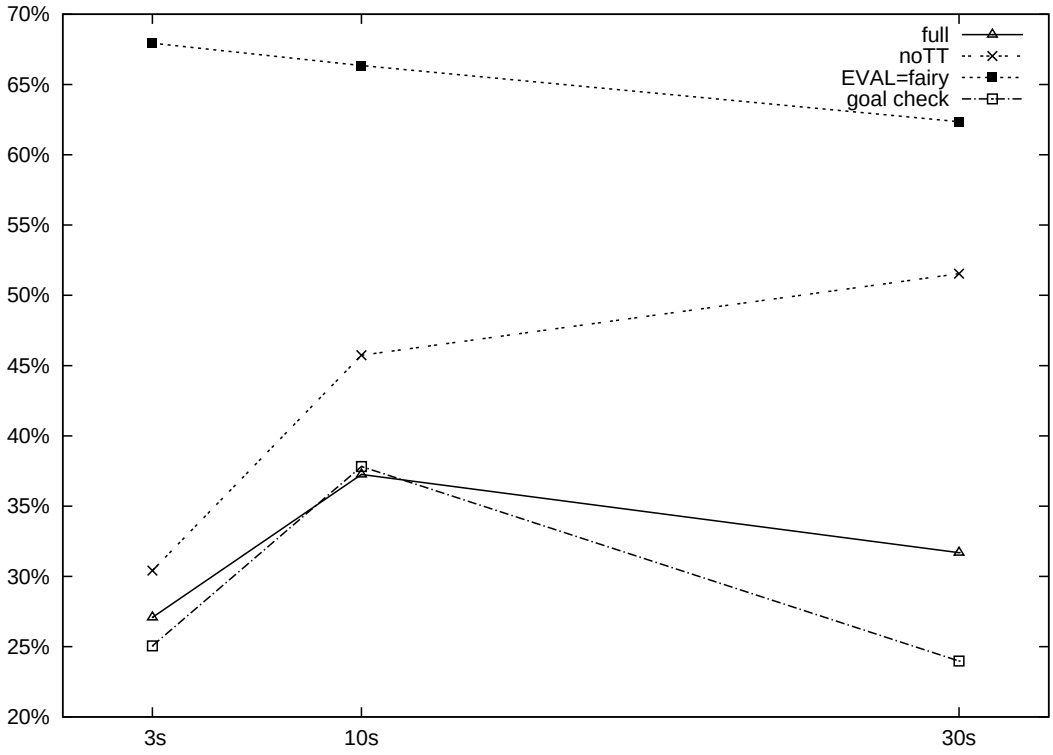


Figure 6.2: Comparing MCTS's and AlphaBeta's general capabilities

In this comparison, it is important to mention how using Fairy's evaluation function affects bots strength. We performed another tests with 10 seconds per turn time limit, in each either AlphaBeta or MCTS version of our bot uses Fairy's evaluation function. Again the information about winning ratio is shown from MCTS's point of view:

AlphaBeta with Fairy's evaluation function vs full MCTS:	41.51%
full AlphaBeta vs MCTS with Fairy's evaluation function:	20.12%

This means that using Fairy's evaluation is making our bots weaker. Therefore we can conclude from the graph in Figure 6.2 that MCTS behaves better with worse evaluation functions compared to AlphaBeta.

From Figure 6.2 we see that MCTS without Transposition Tables defeats AlphaBeta more likely with bigger time limits. A possible explanation is that either Transposition Tables are far more important in AlphaBeta or using History Heuristic in MCTS compensates the loss of Transposition Tables.

At the end of testing process we extended our evaluation function with Kozelek's goal check from his bot Akimot. From Figure 6.2 we see that using goal check helped AlphaBeta a lot. Very likely, this corresponds to the fact, that our AlphaBeta search in 30 seconds per turn explore minimax tree to depth of eight steps more often.

For the next tests we changed the parameters. Both algorithms used all extensions (without goal check), and time limit per turn was set to 15 seconds.

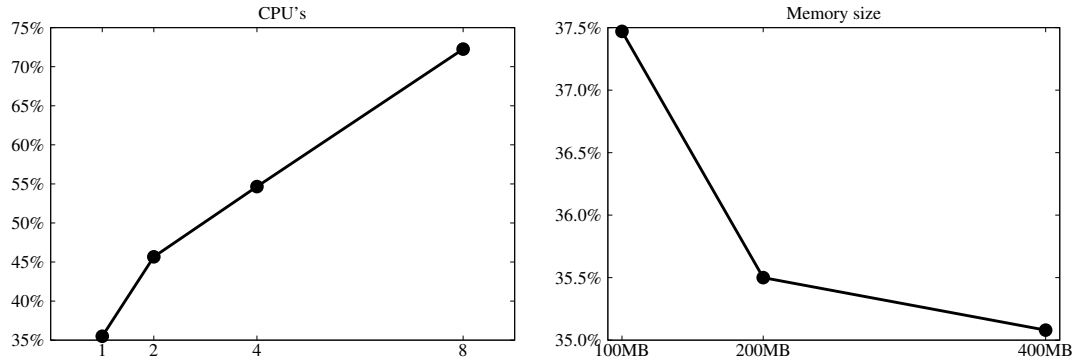


Figure 6.3: Comparison using parallelization or different memory size

Figure 6.3 displays how the winning ratio MCTS AlphaBeta search changed with increasing number of CPU's or amount of memory. The CPU's graph represents how the ratio changed with one, two, four or eight CPU's and the size of Transposition Tables fixed to 200 MB. In the Memory size graph it is shown how the ratio changed with a size limit of Transposition Tables set to 100 MB, 200 MB or 400 MB and the number of CPU's set to one.

For better explanation of the CPU's graph from Figure 6.3 we looked how the number of iterations of MCTS algorithm and the depth of AlphaBeta search in steps changed with added CPU's:

Cores:	MCTS iterations:	Average depth of the AlphaBeta search in steps:
1	4566	6.9
2	8370	8.3
4	16161	8.8
8	22083	8.8

Possible explanations of graph in 6.3 could be following:

- The results from CPU's graph could be disadvantageous for the AlphaBeta search due to some artifact of the implementation.
- AlphaBeta uses mutexes to lock global Transposition Tables during insertions. MCTS uses mutexes for nodes, History heuristic table and Transposition Tables locking. However MCTS spends most of the time in playouts by generating and evaluating steps so it is not affected by the lockings so much (See Appendix A for detailed statistics).
- With limited memory resources (the 200 MB memory limit for Transposition Tables in our case) having more CPU's becomes superior.
- Light static evaluate function helps a lot with increasing number of CPU's.

- For the AlphaBeta version of our bot, having more than 200 MB for Transposition Tables with 15 seconds per turn time limit does not improve the quality of the bot so much.
- The previous conclusion about AlphaBeta's memory needs corresponds with these results.

7. Conclusion

Writing two programs at once turns out to be more difficult than we thought. In our development we have not focussed on time optimisations, sophisticated goal check and trap control in position evaluation. To handle these three topics is considered as the most difficult in Arimaa bot development.

We developed our easily extendable game playing engine named Rabbocop with AlphaBeta search and Monte Carlo Tree Search. A variety of extensions were tried, we took an inspiration from algorithms well known in Chess, Go and Arimaa bot programming. We performed many tests in order to explore how used optimisations help to improve the bots quality and how the used algorithms behave with some restrictions or on better hardware.

From our tests it seems that MCTS behaves much better with less sophisticated evaluation function and for AlphaBeta it is more important to have strong evaluation function. The step evaluation with Best-of-N heuristic promises big importance with increasing power of computers and therefore it is a perfect place for another improvement.

Maybe MCTS will be successful if we let the static evaluation function to be as simple as possible (for example by evaluating just material advantage), and if the domain knowledge will be applied in UCT tree or in playouts. We believe that using more sophisticated board representation would help MCTS algorithm against AlphaBeta, because the cost of managing clever representation returns with more times you use the clever representation, which dominates in MCTS. (From one position or similar to it MCTS in playouts often generates all possible moves more than once in contrast to AlphaBeta).

Using MCTS as we had or as Kozelek provided is not enough for bot programming in Arimaa. The new insights are needed for MCTS algorithm to be competitive with today's Arimaa bots. UCB1 formula itself or UCT algorithm could be valuable in some combination of AlphaBeta and MCTS algorithms. It is possible that in the future, we will see UCT oriented algorithms which after node expansion makes short tactical (AlphaBeta) lookahead.

7.1 Further work

A proceeding work could cover the following problems:

1. Find way how to implement some kind of Quiescence search with goal check and trap control to MCTS and compare with similar approach in AlphaBeta.

2. Optimise playouts using a more sophisticated way how to generate steps. Interesting ideas are described in Zhong's work [6].

3. Try pattern heuristics in playout generation (idea taken from [14, 15]).

There is still long way to go until our bots became competitive with today's best engines. Their quality can be improved in many ways, for example by:

1. Creating better evaluation function.
2. Trying progressive pruning methods from [7, 10].
3. Using more optimised data structures and random number generator.
4. Improving step-evaluation function.

Literature

[1] KOZELEK, Tomáš. *MCTS methods in the game of Arimaa*. Master's thesis. Charles University in Prague, 2009.

[2] COX, Christ-Jan. *Analysis and Implementation of the Game Arimaa*. Master Thesis. Universiteit Maastricht, 2006.

[3] Arimaa homepage. `http://arimaa.com/`.

[4] SYED, Omar, and SYED, Aamir. *Arimaa: A New Game Designed to be Difficult for Computers*. Journal of the International Computer Games Association, 26(2):138–139, 2003.

[5] KNUTH, Donald E., and MOORE, R. W.. *An Analysis of Alpha-Beta Pruning*. Artificial Intelligence Vol. 6, No. 4: 293–326, 1975. Reprinted in Selected Papers on Analysis of Algorithms by Donald E. Knuth. Addison-Wesley, 2000. ISBN: 1-57585-211-5.

[6] ZHONG, Haizhi. *Building a Strong Arimaa-playing Program*. Master Thesis. University of Alberta, 2005.

[7] CHASLOT, Guillaume, WINANDS, Mark, HERIK, Jaap H. van den, UITERWIJK, Jos, and BOUZY, Bruno. *Progressive strategies for Monte-Carlo tree search*. In Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session, 2007.

[8] BERND, Brugmann. *Monte carlo go*. Technical report, 1993.

[9] METROPOLIS, N. *The beginning of the Monte Carlo method*. Los Alamos Science (1987 Special Issue dedicated to Stanisław Ulam): 125–130, 1987.

[10] BOUZY, B., HELMSTETTER, B. *Monte-Carlo Go Development*. Advances in Computer Games 10, 2003.

[11] GELLY, Sylvain, WANG, Yizao. *Exploration exploitation in Go: UCT for Monte-Carlo Go*. University of Paris-Sud, 2006.

[12] AURER, Peter , CESA-BIANCHI, Nicol', and FISCHER, Paul. *Finite-time analysis of the o multiarmed bandit problem*. Mach. Learn., 47(2–3):235–256, 2002.

[13] DRAKE, Peter, and UURTAMO, Steve. *Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go*? In Proceedings of the 3rd North American Game-On Conference, 2007.

[14] GELLY, Sylvain, WANG, Yizao, MUNOS, R'mi, and TEYTAUD, Olivier. *Modification of UCT with patterns in Monte-Carlo Go*. Technical Report 6062, INRIA, France, November 2006.

[15] TRIPPEN, Gerhard. *Plans, Patterns and Move Categories Guiding a Highly Selective Search*. The University of British Columbia, 2009.

[16] FOTLAND, David. *Building a World Champion Arimaa Program*. CG 2004, LNCS 3846 (2006) 175–186.

[17] CARLINI, Stefano. *Arimaa: from rules to bitboard analysys*. Knowledge Representation Thesis. University of Modena and Reggio Emilia, 2008.

[18] KOCSIS, Levente, SZEPESVÁR, Csaba, and WILLEMSON, Jan. *Improved Monte-Carlo Search*. University of Tartu, 2006.

[19] SHAMS, Reza, KAINDL, Hermann, and HORACEK, Helmut. *Using aspiration windows for minimax algorithms*. In Procs. 8th Int. Joint Conf. on Art. Intell, 192–197, Kaufman, 1991.

Glossary

MCTS Monte Carlo Tree Search

UCB Upper Confidence Bounds

UCT Upper Confidence bound to Trees, or UCB applied to minimax trees

Eye in Go is a single empty space inside a group of stones of the same color.

Goalline is the furthest row from the starting rows

Quiescence is an important extension to AlphaBeta search, before evaluating leafs the tactical search is performed looking for easy trapping or scoring goal.

Goal check is an important extension of evaluation function or Quiescence search. Its only purpose is to check whether any player can score goal in actual/next turn.

Playout is the Simulation part of the MCTS.

A. Appendix 1

In our development we used GHC as a Haskell compiler. The following statistics were taken by compiling bots with `-prof -auto` options and ran with `+RTS -p` parameters.

Statistics of the AlphaBeta search

COST CENTRE	MODULE	%time	%alloc
makeStep	BitRepresentation	24.0	34.6
alphaBeta	AlphaBeta	15.6	15.7
eval	BitEval	14.8	3.5
getHash	Hash	9.1	2.9
addHash	Hash	7.2	11.2
isForbidden	BitEval	6.9	3.0
generatePiecesSteps	BitRepresentation	5.7	9.3
bits	MyBits	3.5	6.7
generateMoveable	BitRepresentation	3.0	4.5
newHTables	AlphaBeta	2.2	1.6
stepInMove	BitRepresentation	1.7	3.1
isEnd	BitRepresentation	1.5	0.2

Statistics of the Monte Carlo Tree Search

COST CENTRE	MODULE	%time	%alloc
getValueByMC	MonteCarloEval	23.2	27.1
generatePiecesSteps	BitRepresentation	21.5	21.8
makeStep	BitRepresentation	15.0	17.1
generateMoveable	BitRepresentation	13.3	11.1
evalStep	Eval	7.8	4.6
bits	MyBits	5.7	13.0
isEnd	BitRepresentation	3.3	0.1
stepToInt	BitRepresentation	1.4	0.3
eval	Eval	1.0	0.2

In both tables are functions shown with their names and names of modules in which they are placed. It is important to mention that if a function is called, statistics of that function are not added to our cost centre from which the function was called. This is important for a frequent calling generatePiecesSteps, makeStep and generateMoveable from getValueByMC.

B. Appendix 2 - User documentation

We provide source codes of bots and all small tools we developed during our work. All is available in `http://github.com/JackeLee/rabbocop` or on included CD.

B.1 Compiling options

Source codes of programs are saved on CD in `rabbocop` dictionary. We use the make for building our programs. Simple writing `make IterativeAB` or `make MCTS` compiles either AlphaBeta or MCTS in its standart version of the bot.

VERBOSE=n For IterativeAB program, this will print actual best move and score anytime a depth is fully explored. For MCTS it will print actual best move and score after each n iterations of UCT algorithm.

EVAL=fairy Builds program using Fairy's evaluation function.

NULL_MOVE Enables Null moves for AlphaBeta search.

canPass=1 Adds possibility to generate less than 4 moves.

CORES=X Compiles program with the number of available CPU's set to X. (Default is 1) When running program compiled with CORES=X it is necessary to add +RTS -NX as additional arguments.

WINDOW=X It switches on aspiration window option for AlphaBeta algorithm with Window set to X.

noHH=1 Disables history heuristics optimisations for MCTS.

abHH=1 Enables history heuristics optimisations for AlphaBeta.

noHeavyPlayout=1 This disables heuristic in playouts.

PROF=1 Enables profiling for bots.

playAB, playMCTS, or playMatch Starts new game to play by human vs AlphaBeta search, human versus MCTS or AlphaBeta search versus MCT. For the first time it downloads testing and graphical environments for offline matches.